How it works...

The `multiprocessing.Pool` method applies `function_square` to the input element to perform a simple calculation. The total number of parallel processes is four:

```
pool = multiprocessing.Pool(processes=4)
```

The `pool.map` method submits to the process pool as separate tasks

```
pool_outputs = pool.map(function_square, inputs)
```

The parameter `inputs` is a list of integer from `0` to `100`:

```
inputs = list(range(100))
```

The result of the calculation is stored in `pool_outputs`. Then, the final result is printed:

```
print ('Pool    :', pool_outputs)
```

It is important to note that the result of the `pool.map()` method is equivalent to Python's built-in function `map()`, except that the processes run parallelly.

Using the mpi4py Python module

The Python programming language provides a number of MPI modules to write parallel programs. The most interesting of these is the `mpi4py` library. It is constructed on top of the MPI-1/2 specifications and provides an object-oriented interface, which closely follows MPI-2 C++ bindings. A C MPI user could use this module without learning a new interface. Therefore, it is widely used as an almost full package of an MPI library in Python.

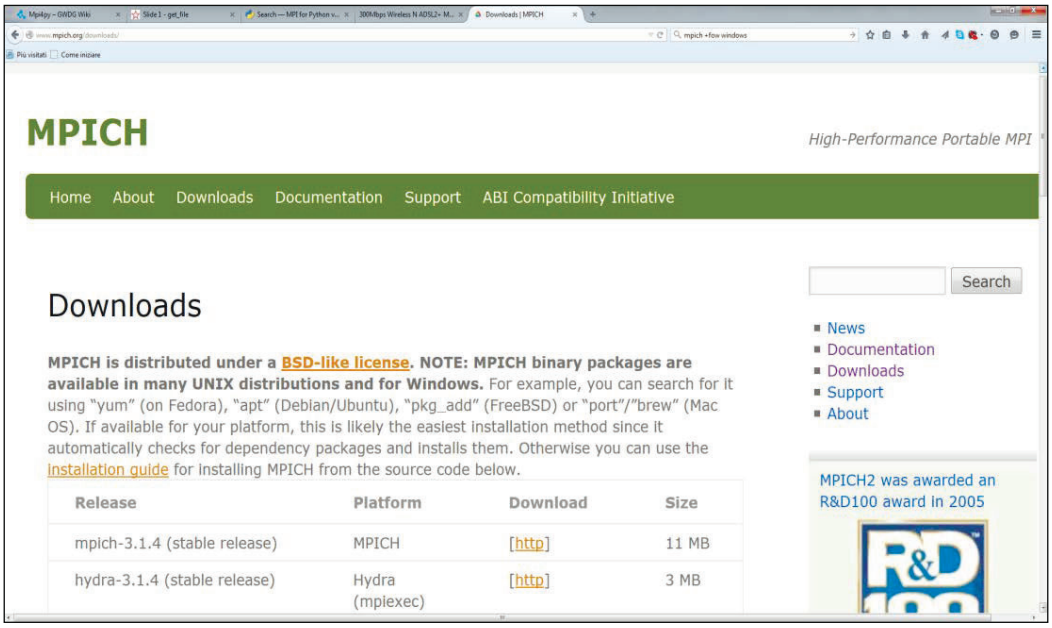
The main applications of the module, which will be described in this chapter, are:

- Point-to-point communication
- Collective communication
- Topologies

Getting ready

The installation procedure of `mpi4py` using a Windows machine is, as follows (for other OS, refer to `http://mpi4py.scipy.org/docs/usrman/install.html#`):

1. Download the MPI software library `mpich` from `http://www.mpich.org/downloads/`.

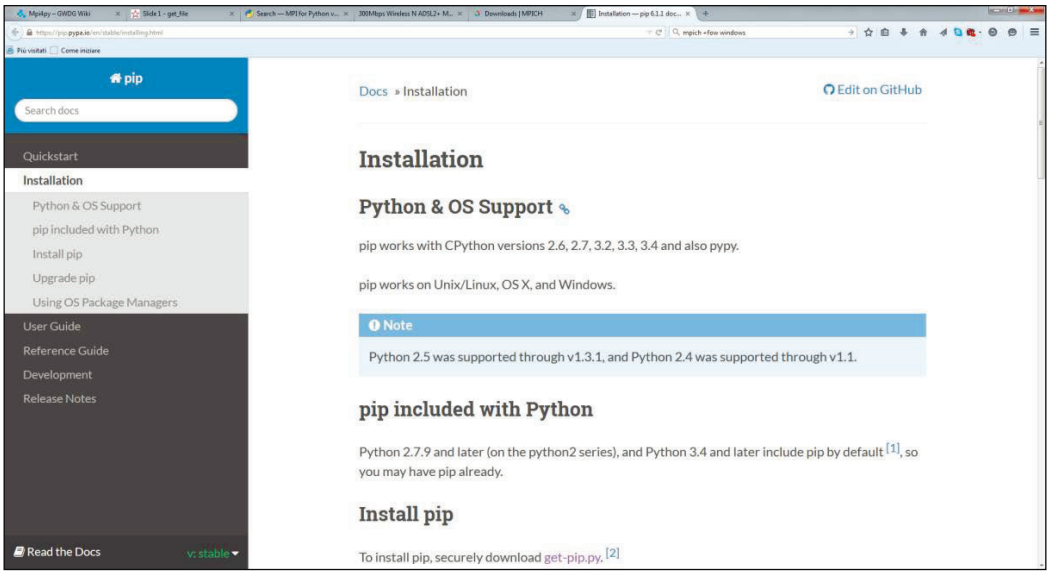


The MPICH download page

2. Open an admin Command Prompt by right-clicking on the command prompt icon and select **Run as administrator**.
3. Run `msiexec /i mpich_installation_file.msi` from the admin Command Prompt to install MPICH2.
4. During the installation, select the option that installs MPICH2 for all users.
5. Run `wmpiconfig` and store the username/password. Use your *real* Windows login name and password.
6. Add `C:\Program Files\MPICH2\bin` to the system path—no need to reboot the machine.
7. Check `smpd` using `smpd -status`. It should return `smpd running on $hostname$`.
8. To test the execution environment, go to the `$MPICHROOT\examples` directory and run `cpi.exe` using `mpiexec -n 4 cpi`.

9. Download the Python installer `pip` from `https://pip.pypa.io/en/stable/installing.html`.

   It will create a `pip.exe` file in the `Scripts` directory of your Python distribution.



The PIP download page

10. Then, from the Command Prompt, type the following to install `mpi4py`:

```
C:> pip install mpi4py
```

How to do it...

Let's start our journey to the MPI library by examining the classic code or a program that prints the phrase "Hello, world!" on each process that is instantiated:

```
#hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print ("hello world from process ", rank)
```

To execute the code, type the following command line:

```
C:> mpiexec  -n 5 python  helloWorld_MPI.py
```

This is the result that we would get after we execute this code:

```
('hello world from process ', 1)
('hello world from process ', 0)
('hello world from process ', 2)
('hello world from process ', 3)
('hello world from process ', 4)
```

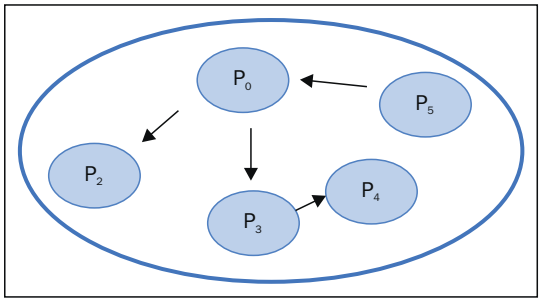
How it works...

In MPI, the processes involved in the execution of a parallel program are identified by a sequence of non-negative integers called ranks. If we have a number *p* of processes that runs a program, the processes will then have a rank that goes from `0` to *p-1*. The function MPI that comes to us to solve this problem has the following function calls:

```
rank = comm.Get_rank()
```

This function returns the rank of the process that called it. The `comm` argument is called a communicator, as it defines its own set of all processes that can communicate together, namely:

```
comm = MPI.COMM_WORLD
```



An example of communication between processes in MPI.COMM_WORLD

There's more...

It should be noted that, for illustration purposes only, the `stdout` output will not always be ordered, as multiple processes can apply at the same time by writing on the screen and the operating system arbitrarily chooses the order. So, we are ready for a fundamental observation: every process involved in the execution of MPI runs the same compiled binary, so each process receives the same instructions to be executed.

Point-to-point communication

One of the most important features among those provided by MPI is the point-to-point communication, which is a mechanism that enables data transmission between two processes: a process receiver, and process sender.

The Python module `mpi4py` enables point-to-point communication via two functions:

- `Comm.Send(data, process_destination)`: This sends data to the destination process identified by its rank in the communicator group
- `Comm.Recv(process_source)`: This receives data from the source process, which is also identified by its rank in the communicator group

The `Comm` parameter, which stands for communicator, defines the group of processes, that may communicate through message passing:

```
comm = MPI.COMM_WORLD
```

How to do it...

In the following example, we show you how to utilize the `comm.send` and `comm.recv` directives to exchange messages between different processes:

```
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)

if rank==0:
    data= 10000000
    destination_process = 4
    comm.send(data,dest=destination_process)
    print ("sending data %s " %data + \
           "to process %d" %destination_process)

if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
    print ("sending data %s :" %data + \
           "to process %d" %destination_process)
```

```
    if rank==4:
        data=comm.recv(source=0)
        print ("data received is = %s" %data)


    if rank==8:
        data1=comm.recv(source=1)
        print ("data1 received is = %s" %data1)
```

To run the script, type the following:

```
C:\>mpiexec -n 9 python pointToPointCommunication.py
```

This is the output that you'll get after you run the script:

```
('my rank is : ', 5)
('my rank is : ', 1)
sending data hello :to process 8
('my rank is : ', 3)
('my rank is : ', 0)
sending data 10000000 to process 4
('my rank is : ', 2)
('my rank is : ', 7)
('my rank is : ', 4)
data received is = 10000000
('my rank is : ', 8)
data1 received is = hello
('my rank is : ', 6)
```

How it works...

We ran the example with a total number of processes equal to nine. So in the communicator group, `comm`, we have nine tasks that can communicate with each other:

```
    comm=MPI.COMM_WORLD
```

Also, to identify a task or processes inside the group, we use their `rank` value:

```
    rank = comm.rank
```

We have two sender processes and two receiver processes.

The process of a rank equal to zero sends numerical data to the receiver process of a rank equal to four:

```
if rank==0:
    data= 10000000
    destination_process = 4
    comm.send(data,dest=destination_process)
```

Similarly, we must specify the receiver process of `rank` equal to four. Also, we note that the `comm.recv` statement must contain as an argument, the rank of the sender process:

```
…
if rank==4:
    data=comm.recv(source=0)
```

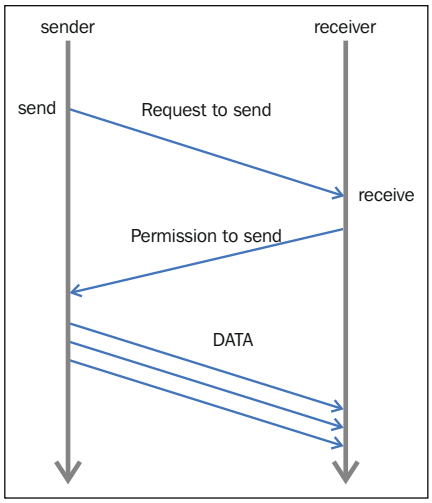
For the other sender and receiver processes, the process of a rank equal to one and the process of a rank equal to eight, respectively, the situation is the same but the only difference is the type of data. In this case, for the sender process, we have a string that is to be sent:

```
if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
```

For the receiver process of a rank equal to eight, the rank of the sender process is pointed out:

```
if rank==8:
    data1=comm.recv(source=1)
```

The following figure summarizes the point-to-point communication protocol in `mpi4py`:



The send/receive transmission protocol

It is a two-step process, consisting of sending some data from one task (**sender**) and of receiving these data by another task (**receiver**). The sending task must specify the data to be sent and their destination (the receiver process), while the receiving task has to specify the source of the message to be received.

There's more...

The `comm.send()` and `comm.recv()` functions are *blocking* functions; they block the caller until the buffered data involved can safely be used. Also in MPI, there are two management methods of sending and receiving messages:

- The buffered mode
- The synchronous mode

In the buffered mode, the flow control returns to the program as soon as the data to be sent has been copied to a buffer. This does not mean that the message is sent or received. In the synchronous mode, however, the function only gets terminated when the corresponding receive function begins receiving the message.

Avoiding deadlock problems

A common problem we face is that of the deadlock. This is a situation where two (or more) processes block each other and wait for the other to perform a certain action that serves to another, and vice versa. The `mpi4py` module doesn't provide any specific functionality to resolve this but only some measures, which the developer must follow to avoid problems of deadlock.

How to do it...

Let's first analyze the following Python code, which will introduce a typical deadlock problem; we have two processes, `rank` equal to one and `rank` equal to five, that communicate which each other and both have the data sender and data receiver functionality:

```
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)
```

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5

    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)

    print ("sending data %s " %data_send + \
           "to process %d" %destination_process)
    print ("data received is = %s" %data_received)



if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1

    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)


    print ("sending data %s :" %data_send + \
           "to process %d" %destination_process)
 print ("data received is = %s" %data_received)
```

How it works...

If we try to run this program (it makes sense to execute it with only two processes), we note that none of the two processes are able to proceed:

```
C:\>mpiexec -n 9 python deadLockProblems.py
('my rank is : ', 8)
('my rank is : ', 3)
('my rank is : ', 2)
('my rank is : ', 7)
('my rank is : ', 0)
('my rank is : ', 4)
('my rank is : ', 6)
```

Both prepare to receive a message from the other and get stuck there. This happens because the function MPI `comm.recv()` as well as the `comm.send()` MPI blocks them. It means that the calling process waits for their completion. As for the `comm.send()` MPI, the completion occurs when the data has been sent and may be overwritten without modifying the message. The completion of the `comm.recv()` MPI, instead, is when the data has been received and can be used. To solve the problem, the first idea that occurs is to invert the `comm.recv()` MPI with the `comm.send()` MPI in this way:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)
```

This solution, however, even if correct from the logical point of view, not always ensures the avoidance of a deadlock. Since the communication is carried out through a buffer, where the `comm.send()` MPI copies the data to be sent, the program runs smoothly only if this buffer is able to hold them all. Otherwise, there is a deadlock: the sender cannot finish sending data because the buffer is committed and the receiver cannot receive data as it is blocked by a `comm.send()` MPI, which is not yet complete. At this point, the solution that allows us to avoid deadlocks is used to swap the sending and receiving functions so as to make them asymmetrical:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)
```

Finally, we get the correct output:

```
C:\>mpiexec -n 9 python deadLockProblems.py

('my rank is : ', 7)
('my rank is : ', 0)
('my rank is : ', 8)
('my rank is : ', 1)
sending data a to process 5
data received is = b
('my rank is : ', 5)
sending data b :to process 1
data received is = a
('my rank is : ', 2)
('my rank is : ', 3)
('my rank is : ', 4)
('my rank is : ', 6)
```

There's more...

The solution to the deadlock is not the only solution. There is, for example, a particular function that unifies the single call that sends a message to a given process and receives another message that comes from another process. This function is called `Sendrecv`:

```
Sendrecv(self, sendbuf, int dest=0, int sendtag=0, recvbuf=None, int
source=0, int recvtag=0, Status status=None)
```

As you can see, the required parameters are the same as the `comm.send()` MPI and the `comm.recv()` MPI. Also, in this case, the function blocks, but compared to the two already seen previously it offers the advantage of leaving the communication subsystem responsible for checking the dependencies between sending and receiving, thus avoiding the deadlock. In this way the code of the previous example becomes as shown:

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    data_received=comm.sendrecv(data_send,dest=destination_process,
                                source =source_process)
if rank==5:
    data_send= "b"
```

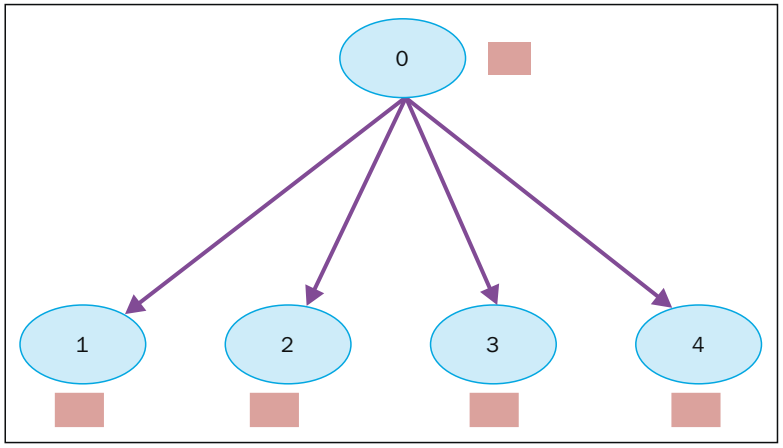
```
destination_process = 1
source_process = 1
data_received=comm.sendrecv(data_send,dest=destination_process,
                            source=source_process)
```

Collective communication using broadcast

During the development of a parallel code, we often find ourselves in the situation where we have to share between multiple processes the value of a certain variable at runtime or certain operations on variables that each process provides (presumably with different values).

To resolve this type of situations, the communication trees are used (for example the process 0 sends data to the processes 1 and 2, which respectively will take care of sending them to the processes 3, 4, 5, and 6, and so on).

Instead, MPI libraries provide functions ideal for the exchange of information or the use of multiple processes that are clearly optimized for the machine in which they are performed.



Broadcasting data from process 0 to processes 1, 2, 3, and 4

A communication method that involves all the processes belonging to a communicator is called a collective communication. Consequently, a collective communication generally involves more than two processes. However, instead of this, we will call the collective communication broadcast, wherein a single process sends the same data to any other process. The `mpi4py` functionalities in the broadcast are offered by the following method:

```
buf = comm.bcast(data_to_share, rank_of_root_process)
```

This function simply sends the information contained in the message process root to every other process that belongs to the `comm` communicator; each process must, however, call it by the same values of `root` and `comm`.

How to do it...

Let's now see an example wherein we've used the broadcast function. We have a root process of `rank` equal to zero that shares its own data, `variable_to_share`, with the other processes defined in the communicator group:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   variable_to_share = 100

else:
   variable_to_share = None

variable_to_share = comm.bcast(variable_to_share, root=0)
print("process = %d" %rank + " variable shared  = %d " \
                     %variable_to_share)
```

The output obtained with a communicator group of ten processes is:

```
C:\>mpiexec -n 10 python broadcast.py
process = 0 variable shared  = 100
process = 8 variable shared  = 100
process = 2 variable shared  = 100
process = 3 variable shared  = 100
process = 4 variable shared  = 100
process = 5 variable shared  = 100
process = 9 variable shared  = 100
process = 6 variable shared  = 100
process = 1 variable shared  = 100
process = 7 variable shared  = 100
```

How it works...

The process root of rank zero instantiates a variable, `variabile_to_share`, equal to `100`. This variable will be shared with the other processes of the communication group:

```
if rank == 0:
   variable_to_share = 100
```

To perform this, we also introduce the broadcasting communication statement:

```
variable_to_share = comm.bcast(variable_to_share, root=0)
```

Here, the parameters in the function are the data to be shared and the root process or main sender process, as denoted in the previous figure. When we run the code, in our case, we have a communication group of ten processes, `variable_to_share` is shared between the others processes in the group. Finally, the `print` statement visualizes the rank of the running process and the value of its variable:

```
print("process = %d" %rank + " variable shared  = %d " \
                     %variable_to_share)
```

There's more...

Collective communication allows simultaneous data transmission between multiple processes in a group. In `mpi4py` the collective communication are provided only in their blocking version (they block the caller method until the buffered data involved can safely be used.)

The most commonly collective operations are:

- Barrier synchronization across the group's processes
- Communication functions:
  - Broadcasting data from one process to all process in the group
  - Gathering data from all process to one process
  - Scattering data from one process to all process
- Reduction operation

Collective communication using scatter

The scatter functionality is very similar to a scatter broadcast but has one major difference, while `comm.bcast` sends the same data to all listening processes, `comm.scatter` can send the chunks of data in an array to different processes. The following figure illustrates the functionality of scatter: